Australian Government
Department of Defence
Defence Science and
Technology Organisation

Tools for Requirements Management: a Comparison of Telelogic DOORS and the HiVe

Tony Cant, Jim McCarthy and Robyn Stanley

Information Networks Division
Defence Science and Technology Organisation

DSTO–GD–0466

ABSTRACT

It is now well-known that a robust and complete requirements management process is of great benefit in the procurement of complex, critical, software-intensive systems. DOORS is a well-established suite of software made by Telelogic, designed to maintain large sets of requirements. The HiVe is a project under development by the TCS Group at DSTO that aims to provide a new approach to the creation of technical documents required in system development. It can be used to formulate, manage, and analyse requirements, and then to develop the system design which satisfies them. While the main focus of each piece of software is different, there is enough overlap that users of DOORS would strongly benefit from use of the HiVe. This report highlights the strengths of both tools, compares their major features, and suggests a number of ways the HiVe and DOORS can interact with one another to benefit the user.

Published by

Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: (08) 8259 5555
Facsimile: (08) 8259 6567

AR No. AR 013-689
July 2006

APPROVED FOR PUBLIC RELEASE

Tools for Requirements Management: a Comparison of Telelogic DOORS and the HiVe

EXECUTIVE SUMMARY

The effective acquisition and integration of computer-based systems – often critical, complex and software-intensive in nature – is a high-risk endeavour. An intensive analysis in the requirements-capture phase is crucial for preventing large cost and schedule overruns.

This report provides advice to the DMO, under Task 04/061 sponsored by the Defence Materiel Organisation (DMO), on the comparison between the HiVe tool – currently being developed by the TCS Group at DSTO – and Telelogic DOORS. The DOORS tool is a mature commercial product to which the DMO has been exposed through its application for managing requirements in various acquisitions. It is highly targeted to this application. The HiVe tool, on the other hand, is under development. Moreover, it takes a rather innovative path in providing highly-structured support for product development: already at the data-entry level it is a sophisticated editor supporting literate documentation production at all levels with the convenience of a central database for project-wide consistency; further, it has a tight and literate coupling to an interactive reasoning environment to facilitate the communication of precise formal design analyses. As such, it is a relatively unknown quantity for the DMO.

In Chapter 1 we discuss systems acquisitions in the Defence environment. We define the concept of *requirements management* and introduce the tools under comparison. The following two chapters elaborate the description of both DOORS and the HiVe software packages, respectively, highlighting the main features that a user of each package will commonly be using. These two chapters can be read independently. The DOORS chapter, being a quick review of an established product, is a summary of the tool features and corresponding user interaction. The HiVe chapter attempts to lay out the ideas which underpin its development and expose the mechanisms and features arising from them which will benefit the user.

The main features of each tool – particularly the cases where one tool has a deficiency that the other addresses – are then compared in Chapter 4. Once the reader is familiar with the feature set of both tools, possible ways the tools can be used to interact with one another are discussed in Chapter 5. Chapter 6 provides a very brief conclusion.

Contents

Figures

Tables

Glossary

Term	Definition
API	Application Programming Interface
Changebars	Visual indication that the contents of a document have changed. Usually a vertical coloured line in the margin alongside the altered text.
CSS	Cascading style sheets
DMO	Defence Materiel Organisation
DOORS	Requirements management software
DXL	Doors eXtension Language. The DOORS inbuilt API that allows DOORS to be extended.
DSTO	Defence Science and Technology Organisation
HIVE	Hierarchical Verification Environment
HTML	Hypertext Markup Language — used for web pages
ID	Identifier
IND	Information Networks Division
Isabelle	Software that provides a generic theorem proving environment
LaTeX	A document typesetting system
MS	Microsoft
NDD	Normative Design Document
OLE	Object Linking and Embedding. A Microsoft approach to linking document types from different applications by embedding a link to the document of another application within a document.
OS	Operating System
PDF	Adobe Portable Document Format
PDFTeX	Software that processes LaTeXsource and produces PDF output
SMTP	Simple Mail Transfer Protocol. Used for sending email.
SoS	Systems of Systems
SRS	Software Requirements Specification
TCS	Trusted Computer Systems Group, DSTO Edinburgh
UML	Unified Modeling Language
Unicode	A universal character encoding standard
Xalan	An XSLT stylesheet processor (that can convert XML documents to other formats)
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction

1.1 The Defence procurement challenge

A central challenge for the Australian Government Department of Defence – and in particular for the Defence Materiel Organisation (DMO) – is the effective acquisition and integration of computer-based systems. Such systems are often critical, complex and software-intensive in nature. Their acquisition is a high-risk endeavour involving a number of development and procurement processes that must take account of the entire lifecycle: from the concept phase; through requirements definition and analysis; through system design, modelling and implementation; to sustainment and finally, disposal. There are many examples of Defence acquisitions where flaws in these processes have led to large cost and schedule overruns.

Clearly, there are huge potential benefits if this acquisition and integration process can be managed in a more rigorous and scientific manner. In particular, improved processes for requirements elucidation, formulation and management are vital for establishing a sound basis for further system engineering development.

In this paper, we discuss the DOORS tool[1] for requirements management, and how this tool is currently used within the DMO. The paper then outlines a new formal-methods-based approach, currently under development by DSTO, that provides tool support (called the HiVe (Hierarchical Verification Environment[2]) for the development of design documentation, with specific advantages for requirements engineering.

1.1.1 Critical Systems

We call a system *critical* if it is the case that certain requirements *must* be met. For example, in a *safety-critical* system we need to ensure that hazardous system states (i.e. those that could lead to an accident) are not reached. In *mission-critical* systems, all the goals and performance requirements critical to the mission must be met. In a *security-critical* system, the system must satisfy a security policy appropriate to its operating environment.

Critical systems need assurance that they will meet their critical requirements; such assurance must be communicated to other parties (such as certifiers).

1.1.2 Complex Systems

Defence systems are also becoming increasingly complex. They may be built up from semi-independent agents responsible for weapons, communications, sensors, control and mobility. These agents will interact with each other (possibly in quite subtle ways), and may be themselves be hierarchical in structure. Their implementation will involve subtle interactions between hardware, software and human components. Complexity may also be manifested by real-time or synchronization issues, as well as stochastic or uncertain behaviour. Large Systems of Systems – such as joint, multi-platform and coalition systems – are also problematic because of the presence of possible unanticipated interfaces and the potential for unexpected emergent properties.

1.1.3 Software-Intensive Systems

Software-intensive systems (those systems that are heavily dependent on software or software-like components such as Field Programmable Gate Arrays) present special challenges: here we need high confidence that the software architecture and design process will achieve the intended result. Assessing and managing the impact of software design changes is very difficult.

1.2 Requirements Management

Requirements management is the process of capturing, analysing and tracking system requirements as they evolve through the system lifecycle. In Defence acquisitions such as SEA 4000 (Air Warfare Destroyer), high-level requirements are usually expressed in the Operational Concept Document (OCD); they are further elaborated in the Function and Performance Specification (FPS) and the System and Sub-System Specification (SSS).

The system requirements must be:

- *valid*, i.e. they must truly reflect user needs;
- *traceable*, so that lower-level requirements are clearly derived from high-level ones;
- *complete*, as omissions will mean that user needs may be ignored;
- *consistent*, as conflicting requirements cannot be satisfied simultaneously;
- *relevant*, as irrelevant requirements could result in inefficient use of resources; and
- *unambiguous* and therefore less likely to lead to misunderstandings.

In a large project, keeping track of all the different requirements, the relationships between them and if they have been met or not can be a daunting task. Tool support is crucial to avert trivial errors introduced through human frailty. The two tools compared in this report are introduced now.

1.2.1 The DOORS Tool

DOORS is part of a commercial suite of requirements management software produced by Telelogic. It is designed to manage large sets of requirements; large projects can have requirements that number in the thousands, with hundreds of users. DOORS also supports concurrent and remote access by many users at once. There are many commercial software packages that offer requirements management functionality; Telelogic DOORS is one of the market leaders. The aim of DOORS is to capture, link, trace, analyze and manage changes to information to ensure a project's compliance with specified requirements and standards.

The DMO is a regular user of DOORS for managing requirements and assessing contracts.

1.2.2 The HiVe Tool

DSTO is undertaking research into methods and tools that support the construction, animation and verification of critical system designs. This work is being sponsored and funded by the DMO. Earlier work culminated in a tool (called DOVE[3]), in which simple systems are analysed using state-machine design models.

A new tool, called the HiVe (Hierarchical Verification Environment) is currently being developed. This tool aims to provide a unified framework in which entire design projects can be captured. The tool will initially support the development of structured technical documentation, providing central management of all for the formal elements required for the design. It will eventually support system modelling using hierarchical dataflow diagrams and state machines, and will also support design verification.

A feature of particular importance is that the HiVe will utilise a formal methods tool. Thus the tool will allow requirements to be formally expressed and – if necessary – provide proof that the design meets these requirements. The term *formal methods* refers to the use of mathematical notation to give precise expression to technical concepts in system engineering.

More specifically, formal methods can be used to:

- model system requirements i.e. convert English language requirements to mathematical statements;
- reason about the properties of system requirements e.g. determine the internal consistency of a set of complex requirements;
- prove that designs and/or code meet their requirements (verification);
- model system design; and
- analyse the meaning (semantics) of software constructs.

The use of formal methods offers a number of key benefits.

Firstly, such methods can lead to a cost reduction in system procurement, by providing support for requirements validation and management, as well as improved understanding of system design and interfaces.

Secondly, formal methods play an important role in providing assurance that critical requirements are met. Indeed they are mandated by a number of safety and security standards). Testing is not sufficient for computer-based systems; it can't be complete because the state space is too big, and it can't be comprehensive because of the discrete state space. A proof tells you how and why a system works, and a formal proof can be machine supported and checked.

Thirdly, formal methods can enhance tool support for the preparation of structured technical documents, making the documentation process more reliable and error-free.

Chapter 2

DOORS overview

DOORS is part of a suite of requirements management software produced by Telelogic. It is designed to manage large sets of requirements; large projects can have requirements that number in the thousands, with hundreds of users. It runs on a variety of platforms, including most versions of MS Windows, HP-UX and Solaris 8 and 9; but not the Macintosh.

Requirements management is the process of capturing and tracking user needs as they change through the development lifecycle. DOORS exists to capture, link, trace, analyze and manage changes to information to ensure a project's compliance with specified requirements and standards. In a large project, keeping track of all the different requirements, the relationships between them and if they have been met or not can be a daunting task. There are many commercial software packages that offer requirements management functionality; Telelogic DOORS is one of the market leaders.

2.1 File and user management

Data in DOORS can be stored in a hierarchical fashion similar to conventional file systems. All data input to DOORS is stored one *object* per line in a spreadsheet like format. Each of these spreadsheets is called a *module* – equivalent to a file in a filesystem view of DOORS – and can contain arbitrary numbers of rows and columns. The basic objects of interest are requirements. Users can arrange objects in a hierarchy and DOORS numbers them accordingly; levels of the hierarchy can be collapsed or expanded to make viewing easier. Each object can have a heading that is displayed on a separate line to the rest of the object data. The hierarchy within the module can be shown in a separate pane to the data in a module for quick navigation, and the objects are identified only by their heading in the navigation pane. The navigation pane can be turned off, leaving only the objects displayed. Figure 2.2 shows the DOORS module view window.

An object can contain more than just a heading and simple text. DOORS allows the adding of extra columns or *attributes*, to objects. The standard DOORS attribute types are integer, real, date, enum and DOORS username. User-defined attribute types are allowed in DOORS, and ranges can be imposed on the type. Some examples are Kg, limited to a positive int; or Deviation, a real ranging from 0.5 to 1.5. The default value of an attribute can also be defined. Attributes can contain formatting such as colours, fonts, bulleting and tables, or even contain charts and graphs. Images can be inserted from files of a limited range of types (.bmp, .wmf, .eps) and *OLE objects*

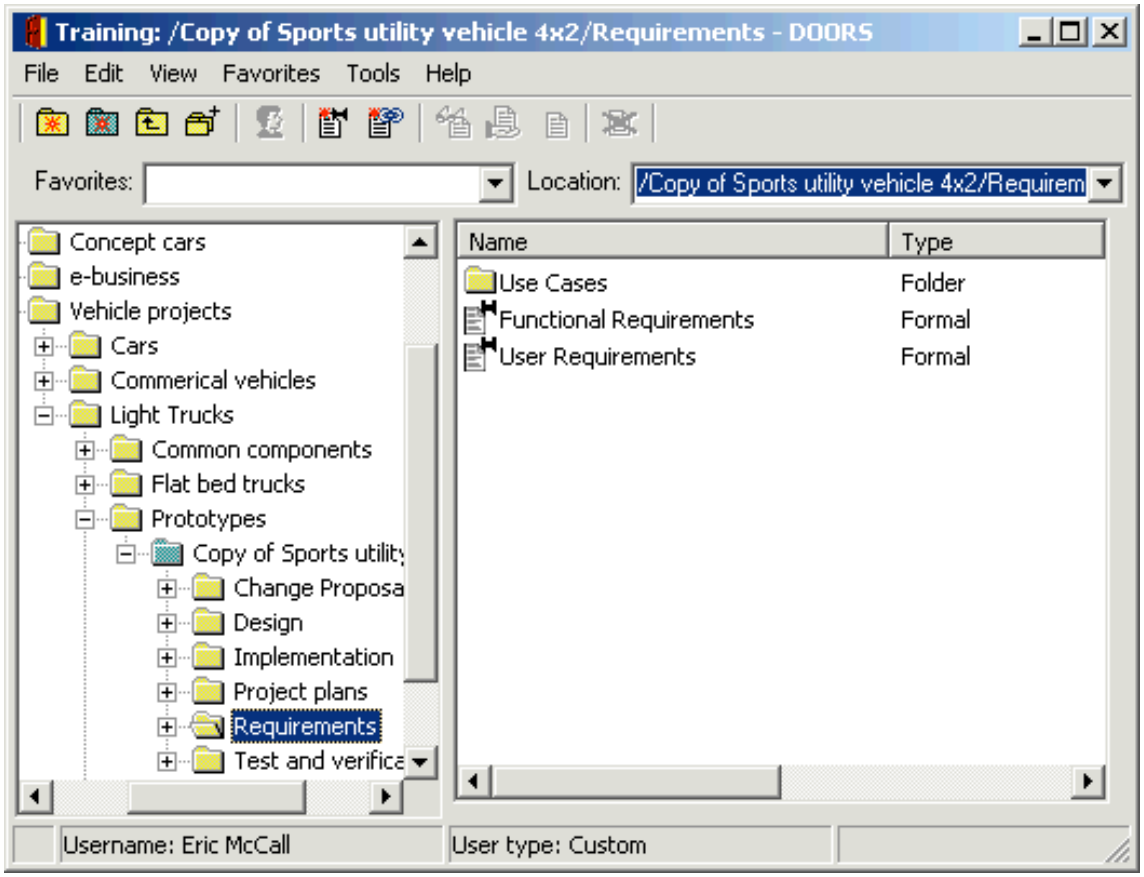

Figure 2.1: *The DOORS split-pane file explorer*

can be inserted in the same way they are inserted into other Windows applications. Values of attributes can be dynamically calculated from the values of other attributes, as can be done in a spreadsheet package.

The DOORS interface is much like common file browsers such as Windows Explorer. DOORS allows users to create folders to store modules in to make navigating a large data set easier. When viewing the DOORS filesystem, the higher-level navigational view of the folder structure is shown in one pane, and the contents of the folders are shown in another pane. This split-viewing system is common to file browsers such as Windows Explorer and document browsers like Adobe Reader. As the DOORS filesystem can get quite large and complex, folders can be bookmarked as favourites by a user to make finding them easier later, in a similar fashion to bookmarking in a web browser. Figure 2.1 shows the main DOORS file browser. Users log into DOORS and can see as much of the DOORS filesystem as their permission level allows. Typical file types that can be seen in the DOORS explorer include modules, descriptive modules (discussed later in Section 2.5) and reports (discussed later in Section 2.4).

DOORS supports multiple users and groups, following a system of access levels and permissions similar to Unix users and groups. DOORS permissions are *read*, *create*, *modify*, *delete* and *administrate* (RCMDA). To group related information, users can make *projects* in the DOORS filesystem. Projects are a special case of a folder, and can be created anywhere in a DOORS

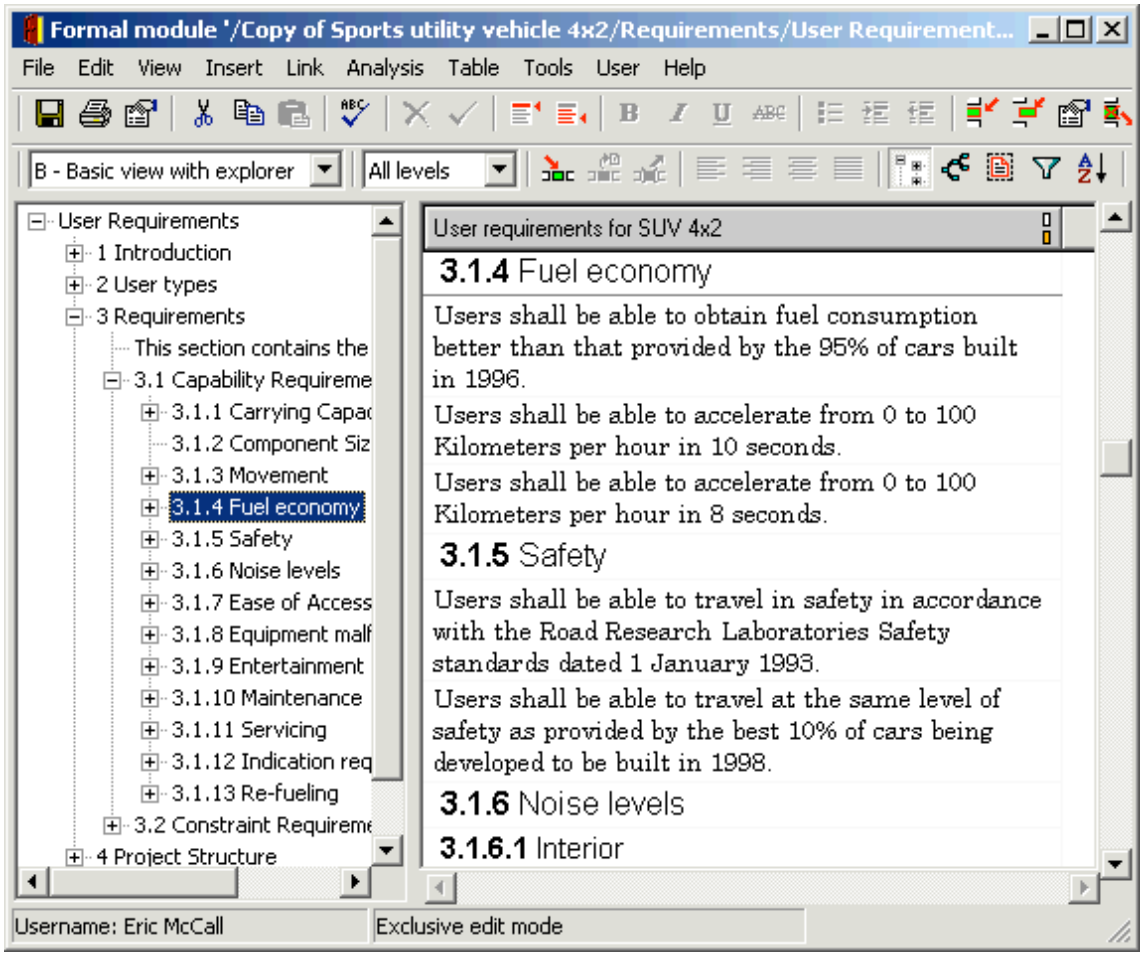

Figure 2.2: *The DOORS split-pane module explorer*

filesystem. Each project has its own set of users, and can contain any number of modules and folders within it. Folders can contain projects, and projects can contain folders. Sets of DOORS users are assigned from the complete population of DOORS users on a project-by-project basis, and can be restricted even further on folders and modules within projects.

With its extensive multi-user support, DOORS can allow several users on different computers to edit parts of the same module at the same time. Any changes made to a module can be flagged visually through the use of *change bars* along the edge of each object, and information about who made the changes is stored in the module history.

Permissions can also be set at the attribute level within a module by a user with the appropriate administrative permissions. This will allow for example certain columns to be hidden or read-only to certain users or groups of users. This is a finer granularity of access levels than the module (or file, or table) level found in other software or at the operating system level. Information about changes to a particular attribute in a module are logged by default to the module history; this logging can be turned off on a per-attribute basis.

DOORS manages editing of requirements by having multiple edit modes and editable sections.

The two standard edit modes are *read-only* and *full edit*. Normally, no two users can open the same module at the same time in full edit mode. However, sometimes it is necessary for this to occur, particularly if one module is very long and multiple users are contributing to it. DOORS allows for this by letting a user (with appropriate permissions) break a module into editable sections. The sections are defined by a depth in the module hierarchy, and should be chosen with as coarse a level of granularity as possible, for performance reasons. If DOORS crashes and a module or editable section has a stale lock, users can unlock their own locks, but only an admin can unlock other users.

2.2 Links

One of the major features of DOORS is its ability to handle links. The idea is that change becomes more traceable if it is known what requirements depend on each other. Links are shown by small arrows at the edge of the main column of rows in a module. The number of links is shown when the mouse is rolled over the links as shown in Figure 2.3. Clicking on the link arrows brings up a menu of links to/from this object, which can be followed to the linked modules. If the link is to a module that is already open, the name of the object that is linked is shown, otherwise "unloaded object" is displayed, as shown in Figure 2.3.

Figure 2.3: *Links in DOORS*

The basic mechanism for creating links is to select requirements, and drag them to the target requirement. Links can also be created in bulk, from the current module to another module, in two different ways. Links can be made between requirements with particular qualities (eg by selecting them based on a regular expression, as discussed later in Section 2.4), and the links can be created between selected requirements. Bulk links can also be created by linking between requirements based on the value of a particular attribute, which effectively links groups of requirements by a key.

DOORS stores links in link modules. There is one link module for every pair of linked modules. Each link module contains linksets, which contain information about all the links between the modules in a particular direction. For modules A and B, their linkset could contain four linksets, representing the links from $A \rightarrow B$, $B \rightarrow A$, $A \rightarrow A$ and $B \rightarrow B$. DOORS normally stores all links that are created in a default linkset for each module. DOORS users can create additional link modules to express different kinds of link relationships, for example to define additional dependencies based on pay centre. Link modules are like normal DOORS modules in that they can

be assigned access permissions, allowing only a particular set of users to see different link sets. Links can also be assigned attributes, adding extra meaning to a link beyond simple direction. These attributes might include who made the link, or a comment about why the link exists.

DOORS provides a number of analysis tools to track requirements that have been affected by changes in the chain of dependencies, or just to track the chain itself. If you are searching on in-links, DOORS calls it an *impact analysis*, and by out-links a *traceability analysis*. A search can be carried out on the default link set, or on a user-defined link set. The number of links to follow can also be restricted.

Links are automatically marked suspect when the source or target is changed, but only if the changed attribute generates module history events – history logging can be turned off on a per-attribute basis. Suspect links can easily be found by filtering or searching for them, and can then be cleared one at a time on inspection. There is also an option to clear all suspect links at once. In the latter case, some links may remain suspect if the module they are in is open for editing at the time of clearing. In a large project, with many different interconnected modules, links could become suspect very quickly. The concept of being able to clear all links to requirements that have been changed at once, without checking them, shows a possible shortfall in DOORS and will be discussed later.

2.3 Baselining and backing up

For every module, DOORS keeps a log file history of actions that users perform, including user logins and sessions as well as changes to data. The module history can be sorted and searched by username or type of change, and is very comprehensive. While this makes it easy to assign accountability and track changes, the sheer volume of information stored can cause performance issues. DOORS opens the log file associated with each module when the module is opened, and this can take some time. Also, the changebars reflect changes since module creation, and carry little meaning if a module is quite old.

Modules and user-defined sets of modules can be “baselined”; i.e., a full snapshot of the module/s at the current moment is stored. Regular baselining will improve performance, as the history logfile and changebars now refer back to the last baselining rather than the full life of the module. Baselines can be electronically signed off, and the signature can include fields such as who signed, and a comment like “All changes approved”. A baseline cannot be edited, but objects within baselined modules can be copied out for use elsewhere. Full modules can also be copied out or used to create templates – which don’t include object text or attributes, only headings – and the links and history of the module will not be copied to the new module.

DOORS includes a module comparison wizard that allows two modules to be compared, either in full or on an attribute-by-attribute basis. It can also be used to compare a module to its baseline, or compare two baselined modules, to quickly see what has changed recently. DOORS simply displays this in the wizard as a list of objects, showing both the original and the new object with the changes highlighted.

Baselines are not backups; DOORS has no explicit backup mechanism. DOORS data can be backed up using conventional filesystem backup tools. DOORS can be used to archive data, but if any modules are open for editing there is a risk that the archive will be incomplete or contain broken links, so it is not recommended to use this technique.

Modules or parts of modules can be exported via a process called partitioning. The exported modules can be imported at another location, edited and then restored back to the original database. Modules can become very large. To avoid the need to take an entire module home each night, DOORS can generate a synch file that can be used to update the original or remote copy. This synch file is effectively just a “diff” that captures the changes made to the module, and DOORS uses it to update the stale module.

DOORS has several built in tools to verify the integrity of a database and to run repairs on it. Database integrity can easily be compromised by crashes which may leave links pointing to objects that don’t exist, so this tool is an essential one. In the worst-case scenario Telelogic advises that technical support be contacted to walk through a database restore.

2.4 Views, filtering, searching and sorting

To help users find what they need more quickly, different views of the requirements can be set up. Attributes can be hidden, different levels of the module hierarchy can be collapsed or expanded by default, and various other features of the screen display can be tweaked to the user’s taste. Users can set up their own custom views, or managers can set up views, for example to hide the cost attribute of a set of requirements from users who are not in the accounting group. All views are set up on a per-module basis.

While views trim modules down largely on a per-column basis, filtering restricts what you see even further, based on far more complicated rules. DOORS offers extensive filtering capability, allowing requirements to be filtered on a particular attribute, or by the status of the links between requirements. The most trivial filter is a text-based search. DOORS allows searches over projects and folders in the same way as file browsers can be asked to search for files, using wildcards such as ? and * and even has the familiar “containing text” option. It is possible to narrow a search by searching for strings occurring in specified attributes, rather than searching over every single attribute in a module.

A number of more advanced filtering options are available. Any attribute of any type can be the subject of a filter, including numeric ranges and date ranges. Full Unix-style regular expressions can be used for text filters. Even the link status and direction of an object can be filtered upon. All these different kinds of filters can be combined by AND, OR and NOT, to produce a fairly advanced filtering capacity. The filter creation dialogue is very similar to that of other applications, where the expression can be typed in manually, or generated using buttons. The results of a filter, or even unfiltered data can be sorted on any attribute in ascending or descending order. The DOORS filter dialogue is shown in Figure 2.4.

The result of a filter can be saved as a report, which can then be printed. DOORS allows some control over the formatting of a report, such as page layout and headers and footers. Reports are stored as files and can be opened from the standard DOORS explorer, and are simply a record of the view or filter that was used to generate the report as well as the formatting information. The data in the report will be re-generated each time. A report generated six months ago will, when re-opened today, not contain six month old data but an up-to-date report.

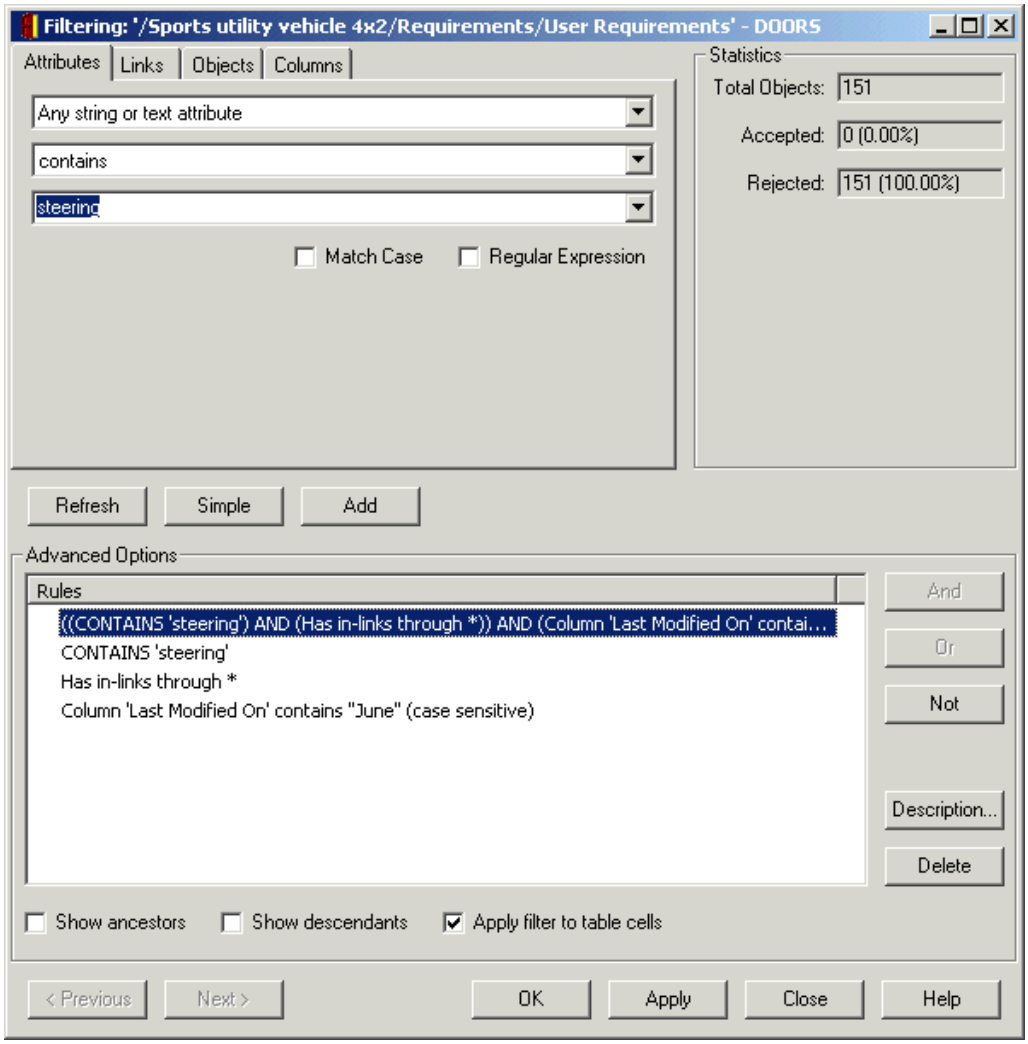

Figure 2.4: *The DOORS filter dialogue*

2.5 Importing and exporting

DOORS provides for a number of ways to import and export data, including the option to write your own import or export routine from scratch. The main file types DOORS deals with are text files, spreadsheets, FrameMaker, rich text, Microsoft Word and Microsoft Project.

2.5.1 Predefined file formats

DOORS by default can import plain text files, and import and export structured documents and spreadsheets. When importing plain text files, DOORS provides a number of different options to help read them correctly, and these can be saved for reuse on other text files in a .map file. Its text file reading module has difficulties with structured text, such as being unable to identify a numbered hierarchy within a text file but will read the same information in a .rtf or Word document.

Spreadsheets are a format that can be both imported and exported in DOORS. In this case "spreadsheet" means a comma or tab separated file, which can be created by applications like Excel. Attributes are made from the columns in the file, and their types derived from the data in the column. The source spreadsheet can be edited outside of DOORS and the DOORS copy updated without having to re-import the entire file from scratch, with just the changes being incorporated. Additionally, small subsets of a module resulting from applying a filter can be exported and edited outside of DOORS, and then that subset can be used to update the DOORS copy. This option is probably provided to avoid working with extremely large files.

For the document-based formats, DOORS maps headings to DOORS object headings, and most other things to object text allowing for formatting like bullets and tables. The export process simply reverses this, but any additional DOORS-specific formatting like links will be lost during the export. DOORS exports FrameMaker files in an incomplete format, meaning DOORS exported files will have to be opened in FrameMaker and saved before they can be re-imported into DOORS. When DOORS imports from a Microsoft Project file, it will also import resources and links; and of course the reverse applies when exporting.

A legacy from earlier versions of DOORS is the support to read text files in all at once and manually mark them up into requirements. The resulting module has only one object, containing the text of the file. This type of module is called a descriptive module. Parts of the text can then be selected and marked up and exported to another DOORS module, leaving the original descriptive module unchanged – descriptive modules cannot be edited. Links are then created between the newly made requirements and the descriptive module. This kind of marking up will only create object headings and object text. Other attributes will need to be added manually if desired.

2.5.2 Custom routines to import and export

As discussed later in Section 2.9, DOORS contains an inbuilt programming language called Doors eXtension Language (DXL) that supports file IO. If the need arises to regularly exchange data with DOORS in a format it doesn't normally support, DXL can be used to add another option to the DOORS menu and a DXL program added to read and/or write this new format.

2.6 Change proposal system

The change proposal system built into DOORS feels at first like an afterthought, but it is clear how it should be used. Anyone who does not have edit access to a DOORS module, but who has read access, may look at a particular requirement – or even a whole project – and want to make a change or suggestion but their permissions do not allow it. In a large project it's hard to walk down the hallway to the person that could make the change, particularly if they are physically somewhere else. As an answer to this situation Telelogic offers the change proposal system.

In essense the change proposal system offers a form-based interface to suggest a change to some part of a project. The module or project, the type of change, the priority and the reason the change is requested are sent to the change proposal team. Once received, the change proposal team can group change proposals together if many similar proposals are received. The team can review proposals, change their status and actually implement the change if they accept the proposal. If

the person who submitted the change has a listed email address and DOORS has an SMTP (mail) server configured, they will be informed via email when the status of their proposal changes.

2.7 Testing

DOORS can be used to track the results of testing. As with many non-core DOORS features, this is not designed for large-scale test tracking, but can be used if your testing requirements are not overly complex. For large scale test tracking, Telelogic recommend using their other products. While they do not specify which product to use, TAU/Tester seems the package of choice listed on their website.

Test definitions can be created on a per-module basis. These define the attributes that need to be recorded for each iteration of a test. Attributes can be chosen from the module attributes, or created from scratch to include, say, who ran the test, the result and other comments. These attributes can be modified later if needed. Each time a test is run and it is time to add the results to DOORS, all the test attributes are duplicated for the new run. This creates a new column for the results, and the user can fill in all the new values. DOORS supplies a tool for later comparison of test runs.

2.8 Other ways of accessing DOORS

Users are not limited to accessing a DOORS database through the DOORS program itself. For lighter work that doesn't need the full functionality of DOORS, or for remote access of a database, Telelogic provides two products that provide additional ways of using DOORS.

2.8.1 DOORSnet

DOORSnet provides a web-based front end to a DOORS database. This removes the need for a client installation of DOORS, and DOORS data can be accessed from anywhere in the world that has a browser and the Internet. Merely installing DOORSnet and setting it up doesn't mean your entire database is accessible from the Internet. DOORS users need to be assigned publishing rights unless only the DOORSnet administrator is going to publish DOORS modules to the web. Then, modules and projects can be published to the web as needed – one project at a time.

Existing DOORS users need to register with DOORSnet before they can use it. A guest user/s can be specified, but if people use the guest account to submit a change proposal they will not get email notification about the status of their change proposal. Guest access to the database can be completely disabled. A license is required to run edit sessions via DOORSnet, as it defaults to read-only access; however the change proposal system is fully functional over the DOORSnet interface to registered users.

Two DOORSnet servers can be run on the same or different computers to increase performance while they serve the same database. One installation of DOORSnet will be the primary one for that database, and all configuring must be done using that server. Multiple DOORS databases cannot be published from the one computer. If multiple DOORS databases need to be published

to the web, multiple DOORSnet servers on separate computers must be used. Every time the published directory of the database is changed, or if the operating system of the server is changed, all registered users must re-register.

There is no inbuilt security for DOORSnet, so if security is a concern the database should be accessed by SSL, using https:// urls rather than http://. And finally, on an aesthetic note, DOORSnet allows a custom header, footer and title to be added to the web pages it generates.

2.8.2 RequireIT

Strictly a Windows program, RequireIT hooks into Microsoft Word and can be used for creating and managing small sets of requirements. It is not recommended for a project containing more than 10 documents; larger projects should use the full-featured DOORS program. RequireIT adds its own menu options and toolbar buttons to Word and uses these to interface with the user. A new or existing document is "marked up" into requirements, and relationships between requirements can be stored as links. Text is marked up by selecting it with the mouse or cursor, and clicking the markup button on the toolbar.

RequireIT stores the attributes of requirements using the hidden text feature of Word, which can be toggled on and off via a button on the toolbar. The attributes can be collapsed or expanded by a small link near the requirement text. The security settings in Word may interfere, as RequireIT is written in Word macros. The hidden text can also cause problems. When using RequireIT, it is safest to show the hidden text before cutting and pasting, as breaking up a block of hidden attribute information will break the underlying DOORS requirement and any links. RequireIT adds some buttons to Word that correctly selects/cuts/copies a requirement based on the cursor position.

Filtering and report creation is supported in RequireIT, again via added menu items and toolbar buttons. Once the appropriate filtering has been applied to a document, a report can be generated as a new document and can be saved or printed in the usual Word fashion. Without creating a report, the results of filtering are simply displayed as numbers of matches, and Word highlighting within the document. Two pre-defined reports are available: Requirements and Traceability. The former generates a two-column Word table suitable for printing, and the latter generates a report on the links in the document.

RequireIT can be used to work with DOORS modules at home, and then the document can be imported into DOORS as a DOORS module. The appropriate licenses need to be held to exchange data between RequireIT and DOORS. Without the licenses the RequireIT documents can only be imported to DOORS as standard Word documents, and will lose attribute and link information.

2.9 Doors Extension Language

One of the less publicised features of DOORS is that it contains an extensive programming language, called the DOORS Extension Language, or DXL. It is DXL that is responsible for many of the more elaborate graphical features that can be placed in attributes, such as graphs. DXL is an interpreted language, with an interpreter and debugger built into the DOORS application.

DXL is very (C/C++)-like in syntax. DOORS ships with a fairly extensive DXL electronic help file, and a library of example programs and routines. The inbuilt DXL editor and interpreter

work in the same way as most interpreters, allowing the user to load a file that's already written or start one from scratch and then edit it in place. The manual doesn't recommend editing large files within DOORS, and suggests editing them elsewhere and loading them into DOORS for testing and debugging. It supports syntax checking, step-by-step execution and other standard features expected of an interpreter.

The familiar C++ features of file access, data types and streams are supported by DXL. In addition, there are functions to create and manipulate GUI objects. DXL can be used to create simple warning dialogue boxes and a number of other pre-defined windows including a text editor. For the more adventurous DXL programmer there is support for building a GUI from scratch, defining all the needed widgets and their placement within the window. There is also support for graphics canvases, allowing a DXL application to draw graphs or charts in a dialog box.

As DXL is fundamentally a DOORS language and any DXL application can only run within the DOORS application, there is extensive support in DXL for DOORS routines. These include defining or changing views, adding items to the DOORS menus, changing the DOORS colour scheme and manipulation of objects and modules within DOORS, including adding and removing attributes. A full API is included for manipulating the DOORS database and linksets. Triggers can be installed into DOORS that cause the launch of a DXL application when a defined event occurs, such as when a particular attribute is modified by the user. Effectively, anything that can be manually done within DOORS can be automated with DXL.

2.10 In summary

Telelogic DOORS is a mature suite of software tools with a strong customer and support base. It is reasonably robust, scales well to hundreds of users and thousands of requirements and can be accessed locally, concurrently, via the internet and through various other means making it highly accessible. It contains an inbuilt API so it can be extended to provide for customised features that the core DOORS tool does not suppply. For simply and efficiently managing large numbers of requirements, DOORS is an ideal choice of application.

Chapter 3

HiVe overview

The HiVe, or *Hierarchical Verification Environment*, is a new approach to the development, evaluation and certification of complex critical systems. It provides a unified, plug-in based framework in which entire sets of design, explanatory and technical documents can be constructed through appropriate tool interactions.

There are two disparate but equally important aims in producing such documentation.

1. To manage the complexity: along with the nontrivial technical specification there are numerous conventions and implementation choices which must be carried consistently through the design. Once the level of complexity is sufficiently daunting, this is best handled by machine – that is, by formal tools and techniques. Such tools are typically pitched at a low level, with specific user input formats to construct appropriate tool scripts. They enforce consistency in typical system development tasks: from mundane or repetitive input and case analysis through to complex reasoning steps.

2. To convince evaluators, system managers and other stakeholders of the correctness of the design. This aim – to present the actual description of the system which provides human understanding – *cannot* be automated and is not naturally attacked at a low level.

A strong focus of the HiVe will be to help the user to achieve these aims.

There are many proof and modelling tools that can be used for low level formal verification. At the same time, there are many document editing systems available to produce system documentation with. However, the two are not easily married, and this lack of a suitable design environment has led to the development of the HiVe. It combines both: it is an *(almost) What You See Is What You Get* (WYSIWYG) document editor that can be extended to interact with external tools, and can incorporate the output from the external tools into its documents in a structured, readable fashion.

The HiVe is being developed by the Trusted Computer Systems Group at DSTO. It is intended to be highly modular, allowing interaction with an arbitrary number of external tools. The modularity of the HiVe is also reflected in the fact that that different application domains will be supported through different *modules* which can utilise a number of tool plug-ins.

Initially the HiVe will interact with a minimal subset of tools that will allow HiVe users to carry out theorem proving in higher-order logic. The first HiVe release will have three components as shown in Figure 3.1.

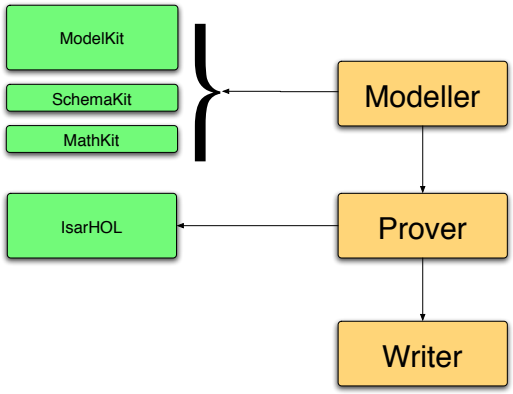

Figure 3.1: *The main components of the* HiVe.

1. The Writer supports the editing of structured technical documents and provides an interface to external tools. It manages all related documents and design constructs within a single project artifact, providing:
   - powerful indexing and structured views, with easy access to all modelling data; and
   - structured (syntax-directed) input of mathematical data and text.
2. The Prover is a plug-in to the Writer which provides theorem-proving support for high assurance (semantic) checking of mathematical models, through:
   - intelligent theory management and proof support; and
   - literate reasoning and theorem proving in Higher-Order Logic.
3. The Modeller is a HiVe module supporting system modelling and verification using the Writer/Prover system. It provides intelligent graphical visualization of design, animation and proof, for tasks such as:
   - formal specification of requirements;
   - sophisticated hierarchical design (for both dataflow and state-machine hierarchies); and
   - reasoning about concurrency, real-time properties and analogue components.

The arrows between these components indicate the dependence described above. The other boxes in the diagram represent the theorem prover tool artifacts which provide the reasoning system required in the indicated components (these will be discussed in more detail in Sections 3.2 and 3.3).

In the following three sections we will consider each of these components in turn. We elaborate on the mechanisms and benefits outlined above. We mainly concentrate on the Writer component, which is the most mature in terms of design and implementation of the tool.

3.1 The WRITER

As discussed, the WRITER supports the preparation of structured technical documents. It assumes an underlying *project* structure, where a project is just equivalent to a folder or directory in a standard file structure, collecting related design constructs and documentation as desired by the user. A given project is built as an extension of existing projects which are said to be *included*. At the bottom of this hierarchy is the core "WRITER" project.

The three principal sub-components of the WRITER are the *Datastore*, the *Normative Design Document* (NDD), and the *Tool Interface* (TI).

1. The Datastore provides a repository for the design artifacts (or *elements*) that the user develops in the project – the corresponding data of all included projects also reposes there. It stores and manages the data, and also provides a palette for insertion into documents as we will describe later.

2. The NDD is a particular document in which the user records the epistemic narrative along with the tool interaction required to produce the design constructs in a literate script.

3. The TI provides an interface between these sub-components and the tools used to process the design constructs.

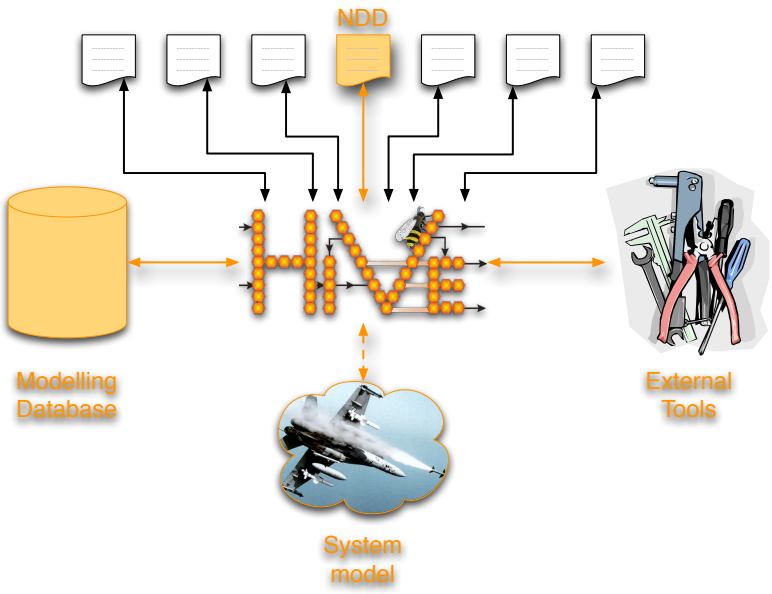

Figure 3.2: *The* HIVE *architecture.*

The interaction between the WRITER sub-components is broadly sketched in Figure 3.2.

- The user writes a linear script in the NDD using the *commands* available in the Datastore. The script may be interspersed with explanatory text which can reference other data from the Datastore.

- the WRITER sends a command to a tool, user-initiated through the NDD script structure; and
- the WRITER accepts response from the tool, distributing data and feedback to the NDD and Datastore as appropriate.

Within a given project there is exactly one Datastore and one NDD. All data in the Datastore is entered through an appropriate command in the NDD. There may be any number of other documents which contain any view of the project – say, pedagogical or summary form, pictorial or slideshow – that the user desires to construct. All occurrences of data in the NDD other than the declaration command, and all occurrences in all other documents generated within the same project, are simply references to the data in the Datastore. Thus, the Datastore is clearly providing a central design model. All documents created within a project will be consistent with this design model and therefore with each other. In the remainder of this section we elaborate on this observation whilst describing the sub-components and tool interface more fully.

3.1.1 Document structure

The WRITER creates documents through the constructors of a formal *document language*. Documents are built up using *blocks* that can be nested within each other, creating a hierarchy. In particular, an element is built as a block in this hierarchy. Consider the example indicated in Figure 3.3, showing a mathematical expression,

$$x + y = 3 \wedge 2x - y = 3\,.$$

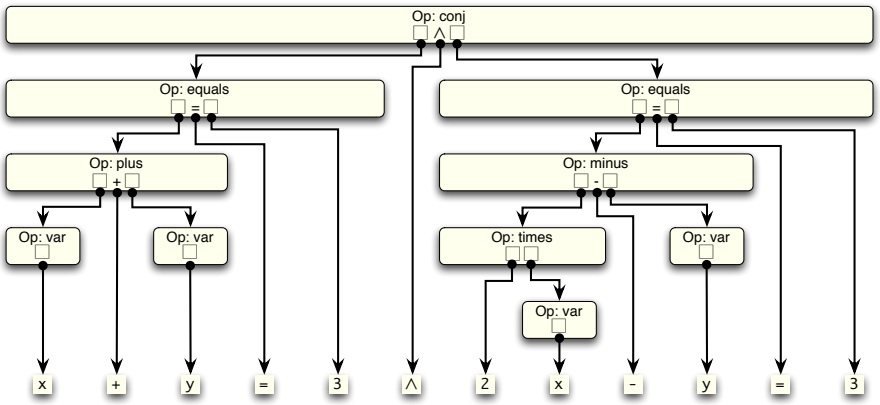

Figure 3.3: *The tree-like operator structure of a mathematical expression.*

The structure of the term is determined by its mathematical operator content: the top level is a conjunction of two subexpressions; each subexpression is an equality; and so on. The block structure branches according to the operator content.

The branching in the above is an example of the mechanism which builds the block hierarchy. More generally, the operators need not just be standard mathematical terms but could act on formatted text – or indeed on whatever the required basic objects are. We have no need to

distinguish at this level. So, in the HiVe, operators are used at all levels in document production: they provide the constructors of an *algebra* with which the document is written. The algebra is specially built for each HiVe module by its developers, and the grammar of the document language is derived from this algebra.

In fact, the algebra is *many-sorted*: blocks are collected into *sorts* similar to the structure of a typed programming language. This provides meta-information about operator arguments through which the Writer can "understand" the content of a HiVe document. There are a variety of ways in which this context-sensitivity is utilised, but here we just consider the example of data entry into documents, as discussed next.

Observe that Figure 3.3 can be read "downwards" as a temporal record of the construction of the term. At each stage the user selects a *dummy block* – i.e., an empty box – and inserts an operator (with further dummy blocks for any arguments) from the Datastore. The dummy block is part of the document language, so each stage in the construction is a legal expression in the document language: the dummy blocks are place-holders for subexpressions. The dummy blocks are filled by other operators, or string/number insertion from the keyboard, or Unicode[4] insertion for pretty-syntax through a font palette. Crucially, any given dummy block will allow only certain sorts for further data entry, restricted by the "many-sorting" of the document language constructor algebra. This mechanism supports the syntax-directed editing function through which the HiVe assists the user with complex documentation and modelling tasks.

Finally, consider the important question of formatting and syntax to display documents in their most natural and informative style appropriate to a given audience. The document language includes mechanisms for modifying the document's appearance. Each block in a document has *attributes*, such as the standard WYSIWYG formatting options. Blocks created inside other blocks inherit the attributes of the parent. The user can customise *presentations* of modelling artifacts – the representation of the way it is displayed – on a per-block basis, and store them in the Datastore. There can be more than one presentation for each artifact, and they are distinguished by assigning a presentation *style*. The style collects presentations together, allowing document modification at a more global level in the same manner that *cascading style sheets* (CSS) can be used in HTML documents for global modification.

The ability to change presentation at a global level can be very useful: e.g., the documents can be "toggled" between presenting mathematical expressions as structured English statements or algebraic terms by the choice of appropriate presentation style. For a requirements document this would allow different audiences to engage with the same consistent content.

It is clear that the Datastore, the document language, and the construction of documents, are tightly coupled in the HiVe solution to the production of technical documentation. Given the above description of the document language we now briefly discuss the Writer sub-components.

3.1.2 The Datastore

A given project has one Datastore, or repository, and data can be both imported from and exported to the Datastore of other projects. It is a standard database in that it is a collection of tables. In particular, all elements are collected in one table which records their result sort and the sorts of their operator arguments (if any). The other table core to the Writer is that of the elements' presentations for screen or hardcopy. Beyond these, the required tables are determined

by the information generated in a given HIVE module or tool plug-in. Each row of a table records information about exactly one element, and is referred to as a *fact* about that element. All data in the tables is structured text, and a given column has a fixed sort.

The Datastore has two functions which are crucial to the user interaction with the WRITER.

1. Data view: the default view of the database will be "indexed" through a browser-type listing which will be organised hierarchically by an internal namespace. Data can be grouped via searches and user-defined or built-in filters formed from database queries. In this way the user can introduce "folders" containing customised collections of elements for convenience.

2. Data input: data can be entered into HIVE documents directly from the Datastore, through drag-and-drop into dummy blocks as discussed in the previous subsection. In particular, each table has a constructor which declares new entries for it and a constructor for displaying a given row. The user can convert folders of elements to single-click palettes for faster insertion.

All such data insertions are references to the corresponding database entry. It is this notion of central storage that makes the HIVE particularly effective at handling multiple documents that are likely to share data, such as different technical documents about the one project. Thus, for example, a change to a highly-used element like "Company name" necessitates only one change in the Datastore, rather than a global search-and-replace in every single document in a project.

The standard interaction utilising this syntax-directed mechanism is that *any* element (formatting, mathematical, tool command, etc) is inserted into a document using the appropriate table's constructor. The interaction is further optimised by utilising the context-sensitivity of the structured document language: when the user selects a dummy block while constructing structured text a simple database query limits the data view to the appropriate sort. Given the often arcane syntax for the objects and commands of external tools it is expected that this syntax-direction will be of use to all users – and of particular use for inexperienced users. It is planned that expert users will be able to bypass the syntax direction where desired by inputting expressions directly from the keyboard into the appropriate dummy block. The input is then parsed into its appropriate form for display in the document and database.

3.1.3 The Normative Design Document

A given project contains exactly one NDD. The NDD is a structured document in the block hierarchy. The NDD is special, however: it is a (literately programmed) script through which the user controls all interactions with tools. In particular, therefore, all Datastore table entries (beyond that of included projects) are entered into the current project through the NDD. This uniformity of tool interaction is supported by the concept of *commands*.

Commands are all of sort *command* – a special sort of structured text. So, for example, a fact in a Datastore table is constructed by an element which produces a command result sort from arguments whose sorts are just those of the table columns. This command is inserted in the NDD to initiate the appropriate tool action. Thus, there is a precise correspondence between each entry of the current projects data in the Datastore tables and a subscript of the commands in the NDD. Moreover, *any* tool command – say, a proof step in a theorem prover tool – will be constructed

similarly. Each command will be sent to the appropriate tool corresponding to the given table – as discussed in the next subsection.

To allow the user to keep contextual focus whilst controlling several tools, the NDD is a *linear* script. Note that indeed it must be constructed in an ordered fashion as determined by the tools it is controlling; e.g., a theorem prover tool will not accept constants which have not been declared at an earlier stage.

Finally, since commands are a particular sort – and so easily recognised in structured text – the linear command script can be embedded in arbitrary pedagogical development of structured text. Thus indeed it will be a literate exposition of the entire project construction.

3.1.4 The Tool Interface

The HiVe can potentially interface to a broad range of external tools. Just a brief list would include: theorem provers and model checkers; algebraic analysis tools and numerical analysis tools; system simulation tools; programming support tools; drawing tools; project management tools; requirements analysis tools; and version control tools. The population of the core tables (of elements and their presentations) are considered an interaction with the core Writer tool.

The HiVe assumes that tools apply some function to input data and commands, to produce output data and diagnostics. The inputs to tools, coming from the NDD command script and the Datastore, and are sent in the appropriate syntactic form to the tool. Tool output is received by the HiVe and inserted as appropriate into the NDD and Datastore. Error messages are handled and displayed appropriately to the user.

The user controls the tool computation through the NDD. The NDD provides the (linear) program script which sequences the computations to be sent to the tool. The command is initiated by the user through a step-wise application of the NDD script, which must be well-ordered with respect to the tool execution logic. Some tools have a finer command grain than others (batch vs interactive, for example) and both are handled by the HiVe.

As mentioned previously, the HiVe architecture is flexible enough to handle a variety of different external tools by means of plug-ins, where the plug-in encapsulates the operator syntax of the tool it interfaces with, and disambiguates identical syntax that has different meaning in different tools. Each plug-in has an initial set of Datastore tables and constructors for its tool, and may add additional GUI elements such as buttons and menu items to allow tool-specific commands to be run from within the HiVe.

3.2 The Prover

The Prover will be a HiVe plug-in for the *Isabelle[5] theorem prover* tool. In this section we briefly discuss the use and choice of Isabelle, what is involved in defining the plug-in, and the benefits of the HiVe in this setting.

3.2.1 Theorem provers

A theorem prover tool provides a safe mechanism for applying inference rules of a given logic to given axioms. A *generic* theorem prover has a core meta-theory in which object logics can be constructed and then extended by the user. Such a tool provides very powerful environment for reasoning in complex or complicated settings without the possibility of making the careless mistakes which are so prevalent (and costly) in human endeavours. The principal application in the HiVe will be for system modelling with high assurance using the Modeller module, but here we just consider the Isabelle tool plug-in for its own sake.

- Isabelle is a *generic* theorem prover.
  - At the core of Isabelle is an abstract datatype for representing the terms and logical propositions to be reasoned about. Associated with this datatype are a small number of trusted functions for constructing logical propositions which are guaranteed to be true. These functions are called the *inference rules* of Isabelle and the true propositions they construct are called *theorems*. The construction of theorems proceeds abstractly – effectively as pattern-matching to the rules – without regard to the meaning (or semantics) of the terms which they manipulate.
  - A recent development for Isabelle is the construction of *Isar*[6] as a "front-end" language. The Isar language provides sophisticated support for the formal definition of mathematical objects and for developing proofs about their properties that are both machine-assisted and human-readable. Its working environment is close to the 'natural' mode of proof for the practising mathematician.
  - Isabelle allows the user to define *theories* which collect named objects and theorems about them. There are a number of *object logics* already constructed this way in the standard distribution. Of principal interest for usual mathematics (especially for system modelling), and the most advanced, is *Higher Order Logic* (HOL)[7]. The user can extend HOL in named theory files as required to model a system of interest. This will be the core inclusion in the Prover module – hence the Isar/HOL box in Figure 3.1.
- Isabelle is an *interactive* proof tool. It implements *proof tactics* which the user must select at each stage of the evolving proof. There are a number of important reasons why an interactive tool is well suited to the task of generic modelling.
  - It is not possible to give a general algorithm which will construct a formal proof for any property. Thus an automatic tool needs to be tightly targeted to a subclass of subgoals. A specific and fixed logical structure is introduced to carry this out. In an interactive proof tool such as Isabelle the logical structure can be extended almost indefinitely, which clearly provides the user with a lot of power.
  - It is not necessarily a good idea to have tactics which make a large change from one proof state to the next, since the user often needs to be able to keep track of the logical changes involved. Automatic proof tools clearly provide an extreme example where the final proof state is very far from the original goal. The machine-generated proof is unlikely to be very intuitive or useful for the user. Moreover, when the automatic proof attempt does not succeed there will be essentially no information to be gained by looking at the current proof state. The user must then devise clever lemmas which the

tool can address successfully on its way towards proving the overall goal. The need to develop this skill is a huge burden on the user. An interactive proof still requires that the small steps be structured towards proving the goal. However, this structuring in itself provides the user with deeper insight into the state machine properties. Failure of the proof attempt can then possibly be correlated with a defect in the machine design.

– At the same time, some convenient level of automation can be implemented in Isabelle by bundling together individual inference rules to make specific tactics.

Of course, automatic theorem provers and model checkers are highly desirable targets for HiVe modules in later releases. The argument here is just that at the first step the interactive prover is much more practical.

3.2.2 The Prover plug-in

As a plug-in for the Isabelle tool, the Prover has four effects on the Writer:

1. it can extend the user interface with menus appropriate to the tool;
2. it extends the DataStore with tables appropriate to the Isar language;
3. it incorporates and manages the files required by the tool, producing them from the natural input to the NDD; and
4. it provides the appropriate code for the Tool Interface which understands how to return from a given Isar command initiated via the user's NDD.

The Prover's graphical interface improves significantly on the basic Isar experience. The Prover will allow significant automation of frequent activities. For example, much of Isar's readability stems from its convention of frequently forcing the user to present the current proof state to the reader. While the benefits to the human reader are obvious, this is clearly a labour intensive process that can be easily automated by extracting the current proof state from Isabelle and importing it into the document at the correct place. Perhaps more usefully, it should be straightforward to provide automated support for common proof strategies, such as induction or equational reasoning.

By storing a simple model of an Isabelle development in the Datastore, the Prover will make it possible for users to inspect and analyse their Isabelle code in new and powerful ways. It will even be possible to include the results of such analyses directly in their documents and to have them automatically update with changes to the code.

3.3 The Modeller

The Modeller is a HiVe module which, using the Prover plug-in, will provide a comprehensive, high-assurance environment for the specification and design of critical engineering projects. This module targets requirements capture and analysis, and design modelling stages of system development which – as we have argued in Chapter 1 – are of critical importance to successful acquisitions projects. An extensive reasoning environment has been developed in Isabelle/Isar to support the HiVe approach to high-assurance developments [8]. A sophisticated GUI front end to this reasoning environment will be developed using the Writer and Prover infrastructure.

3.3.1 Requirements capture and analysis

The MODELLER requirements language features a Z-like [9] schema calculus. The Z schema calculus has proved to be a powerful tool for naming, structuring and reusing complex system requirements. Moreover, for a formal notation, it has a relatively wide user base. As such it is a natural basis from which to build the HIVE requirements language. In the HIVE, the schema calculus is extended (in the Object Z style [10]) with schema types. This brings the the schema calculus benefits of structuring and reuse to the description of complex component hierarchies and is a critical enabler for the treatment of very large systems.

Another notable feature of the HIVE requirements language is an implementation of the Timed Interval Calculus (TIC) which allows the HIVE to treat complex real-time and dynamical systems in a natural manner. TIC was used with great success in the revision of the Prime Item Development Specification (PIDS) for the Nulka decoy [11]. The former Software Verification Research Centre (SVRC) of the University of Queensland, under contract to DSTO, developed a detailed formal specification of the Nulka PIDS, using the TIC [12]. These formal specifications were then translated back into structured English statements. This approach was efficient and led to greatly increased understanding of the requirements. It also highlighted a number of issues with the original (informally expressed) PIDS.

To show some of the strengths of the HIVE requirements language, we consider an example from the Nulka PIDS document: namely, the requirement that the decoy quickly ascend to a safe height above the ship.

We begin by defining a schema to represent the notion of an object's positional track, expressed as map coordinate and height (note the use of an extended typing system: $\mathbb{R} \odot \mathsf{T}$ means a real-valued variable with dimensions of Time, the dimensions made explicit for added fidelity).

$$
\begin{array}{|l}
\hline
\textit{Track} \\
\textit{Height} :: \mathbb{R} \odot \mathsf{T} \rightarrow \mathbb{R} \odot \mathsf{L} \\
\textit{Latitude} :: \mathbb{R} \odot \mathsf{T} \rightarrow \mathbb{R} \\
\textit{Longitude} :: \mathbb{R} \odot \mathsf{T} \rightarrow \mathbb{R} \\
\hline
\end{array}
$$

In the safe flight requirement we need access to the tracks for both the decoy and its ship platform. This can be achieved by declaring variables that have the schema type $\textit{Track}$. In addition, we find it useful to introduce a derived quantity: namely the (ship) relative position track of the decoy.

$$
\begin{array}{|l}
\hline
\textit{ShipDecoy} \\
\textit{Ship} :: \textit{Track} \\
\textit{Decoy}_{abs} :: \textit{Track} \\
\textit{Decoy}_{rel} :: \textit{Track} \\
\hline
\textit{Decoy}_{rel} = \textit{Decoy}_{abs} - \textit{Ship} \\
\hline
\end{array}
$$

The safe flight requirement is formalised first as an abstract schema $\textit{Safe\_Flight}$ on the notion of track. The body of the requirement is expressed as a timed interval condition[1]: namely, that

[1]The interested reader is directed to [12] for a detailed description of the notation. Briefly, a TIC expression P identifies a collection of intervals, called P intervals. The notational conventions used here are: $\langle\phi\rangle$ identifies intervals of time for which ϕ is always true; δ is a reserved symbol returning the length of the current interval; $P \frown Q$ identifies intervals consisting of a P interval immediately followed by a Q interval; and $P \Rrightarrow Q$ says that all P intervals are also Q intervals.

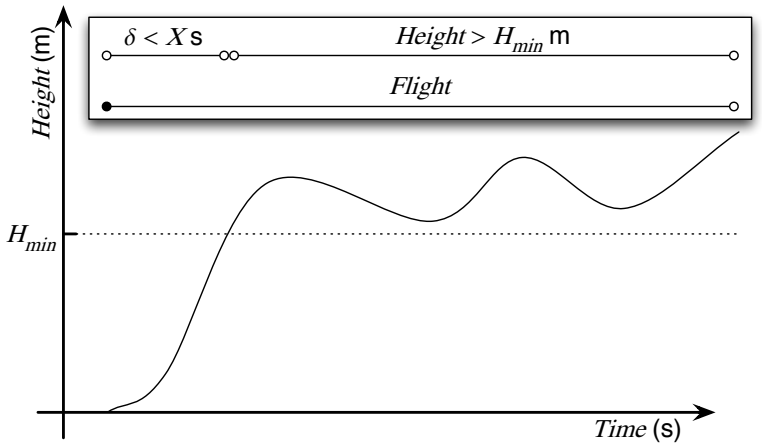

Figure 3.4: *The* $Safe\_Flight$ *condition.*

for any interval of time during which the decoy is in flight mode, the height of the track rises above H_{min}m within Xs and remains so for the duration. The situation described by $Safe\_Flight$, depicted graphically in Figure 3.4.

$$
\begin{array}{|l}
\hline
Safe\_Flight \\
Track \\
\hline
\langle Flight \rangle \Rrightarrow \langle \delta < X\mathrm{s} \rangle \frown \langle Height > H_{min}\mathrm{m} \rangle \\
\hline
\end{array}
$$

Finally, the abstract requirement $Safe\_Flight$ is instantiated to apply to the decoy's relative position track.

$$
\begin{array}{|l}
\hline
Decoy\_Safe\_Flight \\
ShipDecoy \\
\hline
Decoy_{rel}.Safe\_Flight \\
\hline
\end{array}
$$

3.3.2 Horizontal Design Hierarchies

Facilities for imposing hierarchical structure and allowing reuse of common components are critical to the efficient specification and design of complex systems. In addition, evocative visualisations of designs can provide a powerful pedagogic aid. In the HiVe, we adopt the familiar block diagram as the basic tool for system design.

HiVe block diagrams describe simple input/output components. They have well defined input and output interfaces; and allow the unrestricted use of internal interfaces, including feedback loops. A powerful typing discipline is imposed on all interfaces, including the use of structured schema types and polymorphism. Data transformations are effected by computation blocks, considered to evolve with true concurrency unless constrained by some explicit synchronisation

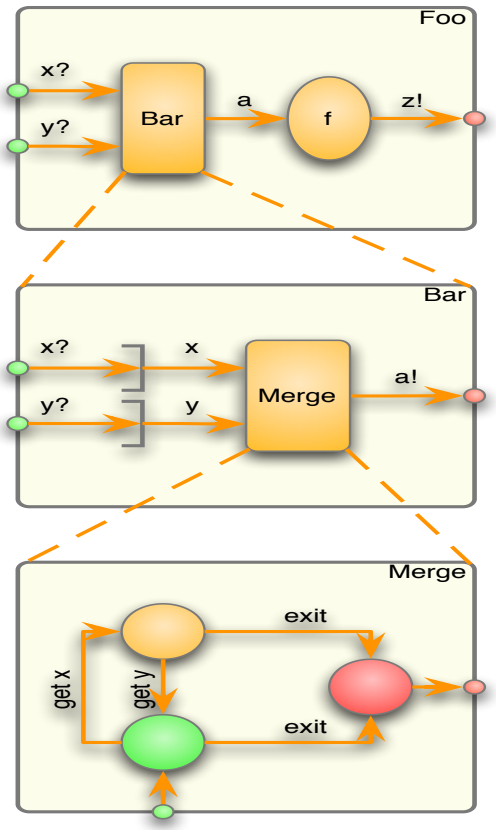

Figure 3.5: *An example horizontal design.*

mechanism. The behaviour of computation blocks may be described by a simple mathematical function, a DOVE-style finite state-machine, or else by a hierarchy of subordinate block diagrams. The resulting sub-component hierarchy is called a *horizontal* design hierarchy since it presents the complete structure of a single design.[2]

Figure 3.5 shows a small example of a block diagram hierarchy. In the top diagram, the Foo component consists of two sub-components: the named Bar component (represented as a rectangular node), which is described as a component hierarchy; and a function component (represented as an oval node) that applies f to a to generate z!. The Bar sub-component diagram is of interest as it demonstrates the reuse of pre-defined components, in this case stream buffer components (represented by vertical tray-like icons). Additionally, the Merge sub-component is described through the use of a state-machine diagram. In the HIVE, the DOVE state machine model is extended with terminal states, to allow a rudimentary form of state-machine hierarchy in which nodes of a parent diagram may represent extended *modes* of computation described by a sub-diagram.

[2]Hierarchies that span a number of levels of the development hierarchy are called vertical designs and are discussed in Section 3.3.3.

3.3.3 Vertical Design Hierarchies

An important mechanism for developing and communicating system assurance is an ability to describe a system at a number of levels of abstraction and to demonstrate correspondence arguments between the different abstractions. For example, Def (Aust) 5679 requires a component-level system design, individual component design models, and component implementation models. Appropriate correspondences must be demonstrated between these abstraction levels, both by testing and reasoning. Such a hierarchy of abstract system descriptions is called a *vertical* design, because it spans a number of abstraction *levels*.

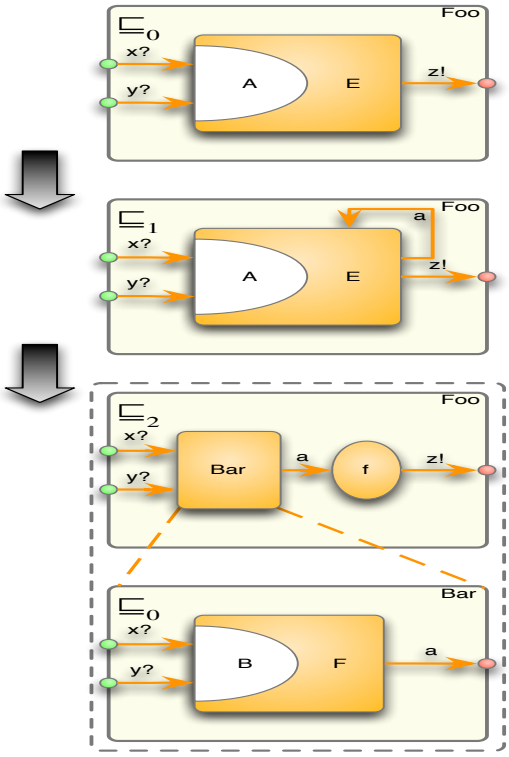

Figure 3.6: *An example vertical design.*

In order to support abstraction hierarchies, the HiVe introduces the *specification* component, inspired by the specification statements of Morgan's Refinement Calculus [13]. Specification components consist of a *commitment* requirement describing the allowed behaviour of the outputs and an *assumption* requirement describing the allowed behaviour of the inputs.

Figure 3.6 shows a vertical system design. The specification components are depicted as rectangular nodes split by an internal arc. The requirements on the concave side of the arc are the assumptions and those on the convex side the commitments. In the $\sqsubseteq_0$ diagram, the Foo is described as a simple specification block. In $\sqsubseteq_1$, this is refined through the introduction of an internal interface a. In $\sqsubseteq_2$, it is further refined by splitting the specification block into a function block and a sub-specification that may be further refined as required.

3.4 Summary

The HiVe is a literate-document WYSIWYG style editor. Documents are built up in the HiVe by a combination of raw typing and syntax-directed insertion-by-reference from the project's Datastore (the database of elements and their attributes). The HiVe supports most standard WYSIWYG editor features, such as fonts, colours, image insertion and most basic formatting.

Tool interaction is controlled by the user through the Normative Design Document. The Datastore/NDD coupling provides a centrally-managed design model; any user documents created within the tool will be consistent with this design model and therefore with each other. Elements that are duplicated between HiVe documents are only duplicated by reference to a central instance in the Datastore, so a change at one central point will be reflected across all documents in a project.

The HiVe can be used to create supporting system development and design documents that can interact with appropriate external tools through a plug-in interface. Initially, the HiVe will include formal design and verification capabilities through a pre-written module utilising a plug-in interface to the Isabelle/Isar environment; additional plug-ins will need to be written by interested parties. Each HiVe plugin will provide the facts and operators to allow the user to create as many new facts, operators and elements as they need from within the HiVe itself. The syntax for any new elements can be chosen convenient or appropriate to the development – rather than being dictated by what is needed to interface with the external tool. The syntax checking will be handled by the plugin for the particular tool.

3.4.1 So what?

The summary hopefully brings out the strengths of the HiVe project, but probably fails to communicate the excitement engendered in such a development environment. There are three major reasons to be excited.

1. Simply, documentation will now be straightforward to construct. It will be standard workflow to build literate documents[3] with fully-captured reasoning at the appropriate stage of the project – when the work is being done.

2. Requirements management, design modelling and verification – crucial to critical system development – will already be fully supported by the initial release described in this report. Notice that this is a complete treatment of the problem; not only can one write down the requirements, but one can then immediately: reason about their consistency; study their completeness; develop their consequences; break them down into component properties; construct implementations at various levels of abstraction; verify that the properties indeed satisfy the requirements; and verify the refinement relation between levels.

3. Summaries of desired information can be presented at all stages in complex reasoning and development activities – even partially automated by the plug-in tuned to a tool's natural mode and to the user's workflow requirements.

Putting these together provides a mechanism for: establishing assurance in a complex system development – or procurement in general – in a timely manner; and, often more importantly,

[3] For an early identification of the importance of literate modelling for requirements analysis, see [14].

transferring the trust to all stakeholders. The immediate pay-off of such an explicit understanding of the system must translate into real benefits from preventing cost and schedule overruns.

The HiVe has a potentially wide range of applications. While initially it is thought to be a tool that helps in the creation of deliverables that satisfy high assurance standards in the development of critical systems, some applications for which the HiVe will be of value include the writing of a mathematical paper, collation of requirements, the description of engineering designs, and the recording of operational mission requirements and implementations for military command and control. With such a wide variety of applications, the HiVe will be of benefit to researchers and developers of a system right through to those who audit and certify the system.

Chapter 4

Comparison of DOORS and the HIVE

DOORS and the HIVE have different objectives, and should not typically be thought of as direct competitors. In this chapter we compare their strengths and weaknesses in system development tasks. There are a number of areas where one is a significant improvement due to the other's limited or lack of functionality there. The results are summarized in Table 4.1.

4.1 Data management

4.1.1 Validity and integrity

Both DOORS and the HIVE store a large amount of data about a given project. There is a large difference in the understanding of the data at entry.

DOORS:

- The user is left to ensure that the entered data is correct. Extra information can be added to user-entered data in the form of attributes. Links can be added between items to show dependencies.
- Features are provided to aid users in maintaining data validity. These tools range from trivial, such as flagging links of changed link items as stale, through to more powerful analysis and sorting tools.
- Both attributes and whole blocks of data can have their access restricted to particular users. Sensitive attributes can be manually set to read-only to all but administrative users. If a change impacts badly on a project, the user responsible can be determined easily.

These validation mechanisms allow data and changes to data to be restricted, tracked, analysed and reported on. These mechanisms are *manual*: there is no sense in which DOORS itself can determine if the change may introduce a problem and, e.g., the user must examine stale links to see if linked items need updating.

All of the DOORS validation features run on meta-data – links – and these features are run (they need not be run at all) by the user well after the data has been entered.

For the user to make use of the mechanisms DOORS provides is a rather large overhead. Indeed, in some sense one would expect linking to be pervasive between development artifacts – so that even the slightest change would produce a large stale link list which the user would find daunting to examine.

DOORS is designed to manage large sets of data with no concern for what the data is, as long as it fits into its table structure – none of its verification tools go to a finer level of detail than a whole cell in a DOORS table.

HiVe:

- The HiVe is designed to ensure that the entered data is valid. A HiVe document is context-sensitive based on the cursor position: only data of the correct sort can be entered there.
- The syntax-direction further helps in explicitly cutting down the available choices. HiVe documents are structured, and are built up in the internal HiVe document language which the syntax-direction presents in a user-friendly palette . This structure is imposed *as the document is created*.
- Feedback from tools is utilised to ensure that the content of HiVe documents remain consistent.
- Data is stored centrally. The typical mode of operation is that: data is modified centrally by the user (in the NDD); the change is then automatically propagated throughout all documents in the project.

The HiVe ensures that anything in its documents is correct according to the external tools it interfaces with, providing a much finer level of control – it allows the user to run verification tools even on a single operator in a mathematical expression embedded in a document.

4.1.2 Linking and embedding

It is worth emphasising the management of changes across a project in the two tools.

DOORS:

- Almost all the representations of relationships between items in DOORS are captured by the hierarchy of stored items and by explicit linking by the user. This can result in a situation where one item is actually identical among many different sets of requirements, and the links simply capture this duplication.
- Items that contain something such as a product name that could be in thousands of places in a DOORS project, and should all be linked to one another for consistency. Changing the product name in one item will not change it in another, they will all need to be changed manually.

HIVE:

- The HIVE has a repository of data elements as well as the WYSIWYG writer from which external tools can be activated. This component bears the largest similarity to DOORS, being effectively a set of tables in rows and columns, attributes and values – like all DOORS modules are.
- Items in the repository are inserted into HIVE documents by *reference*; for example if there are multiple instances of the string "Bob's Hobby Shop" in a HIVE document/s, they all refer back to the one string in the repository. The big advantage of this is that one change to the referenced item in the repository will be immediately reflected in all the instances in the HIVE document/s.
- At one end of the spectrum, the writer of a novel could use this feature to store the names of all the key characters in their book – a simple change in the repository and the lead character becomes Elanora Green instead of Nora Blue. At the other end is, say, the specifications of a software project, where names of files or variable names would be stored in the repository: if they need to be changed then only one instance of the change needs to be made.
- More complicated item formatting or mathematical equations can also be stored in the repository, with one central change being reflected in all HIVE document/s. The repository can store several different formats of display information for one document, allowing for global and consistent application of formatting styles. As the repository can store meta-data about items, such as type and formatting information, this ability of the HIVE is far more than a simple global search-and-replace.

4.1.3 Filtering and sorting

DOORS:

- DOORS is designed to handle extremely large datasets, so it comes equipped with powerful filtering and searching tools.
- The intent of much of the filtering is to find smaller sets of DOORS data to create links between or to create reports from. A secondary purpose would be to find specific data, or to find out who was responsible for a particular item or change to an item.

Without the ability to filter data down to a manageable size, DOORS would be no more useful than a database package without a query language.

HIVE:

- Data sets in the HIVE could also become very large, so the HIVE has a similar filtering capacity to DOORS.
- Filtering in the HIVE is both context-driven and user-defined. That is, the HIVE has the additional capacity to filter by context when inserting into a particular point in a document. The visible items in the repository are filtered out if it would be syntactically incorrect to insert them at the cursor's current location.

- As the HiVe leans heavily on inserting items from a repository of items, the ability to search those items is quite sophisticated.

The level of searchability and filterability in both the HiVe and DOORS is quite similar. Both applications have the ability to store shortcuts to heavily referenced items.

4.2 Report generation

DOORS:

- DOORS can create several different forms of output: simple output to Microsoft Word or other document formats, or in a report format that is designed to be sent straight to the printer.
- DOORS offers minimal control over the look and content of the documents it exports.
- When DOORS creates a report, it simply keeps track of where the data in the report is sourced from, and basic formatting such as headers, footers and page size. A copy of the text of the report is *not* saved by DOORS; the hardcopy sent to the printer will be the only record unless the user makes the effort to baseline the module at that point or export the same data to a Microsoft Word document. The next time the report is opened for any reason, the content of the report will be re-generated from the most recent copy of the source data.

Being predominantly a storage application, DOORS' ability to output information in a format suitable for printing or otherwise distributing in document format is not one of its best features. The relative automation of the report generation may appear superficially attractive, but the lack of any narrative in what is effectively a data dump makes its limitations obvious. A huge amount of *post hoc* work – which would, in a usual workflow, often be very much after the reasoning behind data sets was established – would be required to cut down the table output and insert a narrative.

HiVe:

- The HiVe's emphasis is on producing attractive, well-formatted and readable documents. Moreover, the usual workflow would be to capture these arguments whilst the data is being entered and constructed.
- It allows a high level of customisability, allowing the user to exactly specify the format/presentation of any HiVe element that appears in a document.
- Any document generated within the HiVe can be exported in a format such as Postscript or PDF, which can be printed as well as providing a full-text copy of the HiVe document being exported. These documents can be saved to disk wherever the user wishes, and can be opened later without finding that the data within them has changed, but still reflects the state of the data at the time the document was generated. This behaviour is standard in most document editors.

The HiVe is developed as an elegant system design tool, producing and recording a strong argument for the correctness of the system in a *human readable format*.

4.3 Programming API

DOORS:

DOORS provides a full API called the DOORS eXtension language, or DXL. It allows DOORS to be extended to handle new file types, to provide calculations on the fly, or to draw custom graphics; and many other things. Effectively DXL allows third party plugins to be written for DOORS. This full extensibility offered by DOORS allows an enormous amount of flexibility over and above the basic DOORS functionality. The concept of allowing a large program to be extended is a common one adopted by many software packages.

HiVe:

The HiVe is specifically built to be able to be extended to interface with arbitrary external tools, allowing it to be used seamlessly to edit documents describing a system in any language for which a HiVe plug-in exists.

4.4 Summary

In Table 4.1 the reader will find the results of the the previous sections tabulated for easy comparison. In the remainder of this summary section we highlight the directly contrasting features, concluding with a brief "thought-example" which compares approaches on a requirements-capture problem of interest.

DOORS is primarily designed to capture requirements of a system: long lists of items with additional information recorded about them. DOORS allows users to capture the relationships between them requirements, but this is a manual process that the user must explicitly choose to do. The HiVe is primarily designed to manage elements, produce technical documents about and formally prove properties about the system it is being used to document. The HiVe makes use of third party tools to make these formal decisions.

DOORS is designed for very large datasets, multiple concurrent users and distributed access to data in a variety of different ways. The HiVe currently has no such multi-user capability, and does not support distributed access. It is not expected that the HiVe will support multiple users in the first instance, although this feature may be added in a later release. DOORS has its own internal user base and logon protocol, which can be set-up so it corresponds to the Windows user of a particular computer.

DOORS keeps an extensive changelog on every module, recording user sessions, and tracking ownership of change. It has extremely limited version control, allowing "baselining" (see Section 2.3) that captures a snapshot of a module or set of modules at a given time. The HiVe is single-user only and lacks any built-in change tracking and version control, but HiVe files are stored in a format suitable for use with CVS systems. In conjunction with CVS and a multi-user operating system the HiVe mirrors many of the user specific logging features of DOORS by adding full version control, assigns ownership of change and adds rollback capability to the HiVe. Even such a loose coupling of the HiVe and CVS would exceed the version control in DOORS.

DOORS is a database tool which comes with additional features that can be used to capture relationships between and attributes of its basic data elements. The smallest unit in DOORS is a

Feature	DOORS	HiVe
Document production		
Document structuring	No - Hierarchical numbered lists	Yes
Rich text	Yes - per cell	Yes - Full
Mathematical expressions	Yes - as text	Yes - Syntax directed editing
Portable documents	No - MS Word as default, other formats can be added via DXL	Yes - PDF, and other formats can be added via plugins
Printing documents	Auto-generated, straight to printer	Yes - PDF files to print
Faithful to screen	No	Yes
Data management		
Project-wide editing	No	Yes
Datatypes supported	text, date, integer etc	Yes - HOL and any user-defined type
Consistency management	No	Yes
Change propogation	Manual tracing through links	Highly automated
Very large databases	Yes	Yes with standard database tools
Version control	Logging and baselining but not rollback capability	Yes (with CVS)
Multiple users	Yes	Yes (with CVS and multi-user OS)
Data views		
Context sensitive	No	Yes
Data filtering and sorting	Yes	Yes - both programs are equally powerful
Restrict view/access per user/group	Yes	No
Requirements and modelling		
Requirements capture	Yes - textual	Yes
Requirements tracing	Yes - links	Yes
Requirements modelling	No	Yes
Requirements verification	No	Yes
Requirements structuring	Yes - with headings and modules	Yes
Extendability		
Programming API	Yes - DXL	Yes
External program support	Yes - with DXL	Yes - HiVe is a framework designed for extension with plugins. Plugins may already exist (or can be written) to support additional tools.

Table 4.1: *At a glance: comparison of features in DOORS and the* HiVe.

single requirement – which may have additional attributes – that can contain basic data such as dates, numbers, text, links to other requirements, graphs, documents from other applications and pictures. The smallest unit in the HiVe is as small as the user chooses to make it: a single character, a lone mathematical symbol, or a composite expression that is built up from several other smaller sub-expressions. While the HiVe is not as strongly geared to maintaining extremely large stores of requirements and tracking information about them as DOORS is, it is far more flexible in what it can store, and what information can be *derived* from the data it stores.

The only information DOORS can derive from the data it stores are simple summaries about the data; how many links go where, which items have a particular property, how many requirements were altered by a particular user. In many cases, information of this nature is very useful and the market share DOORS enjoys would attest to this. However, DOORS has no ability to reason about the nature of the data itself; the question of whether or not a requirement is valid with respect to other requirements or even to itself is not something that can be answered with DOORS, but can be easily answered in the HiVe. For critical applications that need a high level of assurance that their requirements have been correctly specified, DOORS cannot provide this assurance. The HiVe can.

4.4.1 A thought-example

The Nulka Decoy, discussed in Section 3.3.1, is an example of a project that could be specified both in plain English or in formal mathematics that expresses the same thing more concisely, and can be reasoned with. The benefits of expressing requirements using formal notation are very clear – properties of the requirements can be formally reasoned about, and proofs about such properties can be obtained so that there is no doubt about these properties.

For requirements formulated in either plain English or structured mathematics, requirements are added to DOORS one requirement at a time. Entry of mathematical text will be limited by what font is available to display mathematical characters. Each requirement takes one line in a DOORS module, and can be numbered hierarchically. If there are very many requirements they can be logically broken up into multiple DOORS modules. For each module, additional information can then be added to the requirements by adding extra columns to the module table. If a requirement has some sort of relationship with another requirement that goes beyond a simple hierarchical relationship, links between requirements can be added and annotated accordingly.

Entering a small set of requirements in the HiVe opens up a variety of options. At the simplest level, they can be added to the NDD as free text and formatted as a numbered list. The user can also add them one at a time to the Datastore (and the NDD by extension), one requirement per line of the database, and add a narrative to the NDD. When requirements are specified in formal mathematical syntax, the expressions can be built up in the HiVe using the appropriate grammar. The formal mathematics will thus be entered into the document in a syntactically correct fashion, and can be manipulated from within the HiVe if needed (by an outside tool or during day-to-day editing), while retaining correct syntax.

Even when requirements are specified in a formal, mathematical syntax, they can only be entered into DOORS in their plain-text form. DOORS makes no distinction between datatypes over and above its standard types of string, real, date etc. Mathematical expressions embedded in text are no different to any other unicode symbol embedded in text. So while both applications

can be used to store a requirement expressed in mathematical format, only the HiVe assigns any meaning to this representation – and only the HiVe can be used to send such information to the appropriate verification tool. Formal specifications stored in DOORS would have to be manually entered into a verification tool by the user.

Chapter 5

Tool interactions

Users of both the HiVe and DOORS may wish to export their files and simply convert them to the format of the other application. In this chapter we discuss how this might be done.

5.1 Static interactions

5.1.1 HiVe reading DOORS format

As it is being suggested that the HiVe will be a plugin-based architecture, this option would involve writing a HiVe plugin that recognises files in DOORS file format and adds appropriate options to the HiVe menus to read in DOORS modules, or more likely whole projects in DOORS format.

Module files in DOORS are quite similiar to standard spreadsheet files, and lack the metadata that links provide. Full projects in DOORS contain a full set of module files and the links between them (links cannot be made between modules that are in separate projects). Far more is gained by adding the ability for the HiVe to read full projects than just single module files. Of course, a lot of metadata about the DOORS project and modules would be lost in the conversion to the internal HiVe format, such as its change history, and all user information and permissions. DOORS supports multi-user permissions and the HiVe currently does not.

5.1.2 HiVe exporting to DOORS format

Again, this option would be implemented by a HiVe plugin. References in both ordinary HiVe documents and the NDD would be captured by DOORS links and stored in an appropriately fleshed-out DOORS link module. DOORS allows users to store metadata about links in a link module but doesn't make use of this feature by default; this feature would be one that a HiVe export routine would make good use of.

The exported DOORS modules and link modules would be free of user information and permissions, in much the same way that DOORS modules exported from other installations of DOORS would be free of this information. Since DOORS itself creates modules in this format, they are in a valid format and DOORS could import these modules and their link modules easily.

5.1.3 DOORS reading HiVE format

For those users who regularly use DOORS rather than the HiVE, such as management or other non-authors, a plugin for DOORS could be written that allows DOORS to input a HiVE project. As HiVE projects need at minimum the data repository and NDD to function correctly, the DOORS plugin would input entire HiVE projects. This plugin would be written in the DOORS extension language (DXL)to allow transparent importing of HiVE projects into DOORS. A menu option would be presented to the DOORS user in the normal DOORS file menu, and the HiVE project on disk would remain unchanged. The base file-reading code from within the HiVE itself should port well to DXL, with the conversion to in-memory structures (DOORS objects and links) being DOORS-specific.

This particular option may provide a more accurate conversion than simply exporting a HiVE project to a DOORS module on disk, as the DOORS structures are built up in memory and will inherit user permissions from the current DOORS project. DOORS itself would capture which user imported the data in its changelog.

5.1.4 DOORS exporting to HiVE format

Again, a DXL plugin could be created to write out a currently opened DOORS project in native HiVE format. As discussed in the previous section, an option would be added to the DOORS menu to export to HiVE. The code behind this technique would be very similar to writing out a standard HiVE file, but from DOORS memory structures, not HiVE ones. In all the conversion techniques discussed in this section, modular code and input/output library routines would mean a lot of code could be reused between different format conversion routines.

5.2 Discussion

In the previous section we outlined the four ways that information can be exchanged statically between DOORS and the HiVE by basic file format conversion. The exact conversion between the DOORS and HiVE formats is a technical issue that would need to be investigated further. DOORS supports the addition of plugins via DXL, and there is substantial existing documentation on how to implement the type of plugin that would be needed. The full specification of HiVE file formats is still being finalised, and information about the internal format of DOORS files would need to be obtained.

Full import/export functionality between DOORS and the HiVE would require implementing all four static interchanges. This would be quite some work. One minimal approach which adds value is the suggestion in Section 5.1.4, where DOORS users can export a DOORS project to HiVE project format, allowing DOORS projects to be opened in the HiVE. Of course, this simply introduces a class of “plain text expressions” of requirements which can be formalised as the user desires. At the least, this sets the user up for developing the requirements analysis, and there could be support for ensuring a complete correspondence between formal and plain text expressions. The export routine would need to be run by an administrator.

Equally, a HIVE plugin that reads DOORS projects would achieve a similar result, but may run into access permissions depending on how DOORS implements its user-restricted access, particularly in a distributed environment.

The HIVE is more effective when a document is built up from scratch within the HIVE environment –as we saw in Section 4.4.1. For this, the minimal approach in the opposite direction would be useful – particularly in the case that DOORS projects have been included historically in the workflow of an organisation. This might also be done best on the DOORS side, as described in Section 5.1.3.

Of course, a conversion from another file format, particularly when converting to/from a dissimilar application such as DOORS, will never give completely the same result as using the HIVE from the outset. Document content that could be captured by the HIVE document language and stored in the modelling database will be missed, and stored as narrative text instead.

One could also consider an "on-the-fly" dynamical interaction between the two tools; i.e., to construct a DOORS plugin to the HIVE. This would be a lot of work to implement, and the payoff does not seem to justify it.

Chapter 6

Conclusion

This report has considered the challenge of procuring systems with assured critical capability from the viewpoint of available and desired tool support. It has demonstrated that the HiVe tool adds real value at the front-end of the process – where a proper analysis can save agonies in later development stages. Of particular importance is the requirements management phase of development, where the analysis explores the completeness and consistency of the requirements and subsequent refinements of corresponding system properties to detailed design. Included therein is a mechanism for listing and tracking requirements throughout this activity.

We have described how the DOORS tool has been used to effect for this requirements listing and tracking. The comparison has highlighted certain innovative advantages of the HiVe which, together with the powerful analyses discussed above, will make it an essential tool in requirements management.

Given the wide acceptance of DOORS for requirements tracing, we have also considered ways in which the two tools could interact – not least as a possible mechanism for increasing the target market of the HiVe.

Acknowledgement:

Generous funding support provided by the DMO for the HiVe implementation is gratefully acknowledged.

References

1. Telelogic DOORS. http://www.telelogic.com/products/doorsers/doors/.

2. T. Cant, B. P. Mahony, J. McCarthy, and L. Vu. Hierarchical verification environment. In T. Cant, editor, *Proceedings of the Tenth Australian Workshop Safety Critical Systems and Software*, volume 55 of *Conferences in Research and Practice in Information Technology, ACS*, pages 47–57, 2005.

3. T. Cant, B. P. Mahony, and J. McCarthy. *Design Oriented Verification and Evaluation: The DOVE Project*, 2002. DSTO Research Report DSTO-TR-1349.

4. Unicode. http://www.unicode.org/.

5. L. C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

6. Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

7. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle's Logics: HOL*, 2001. Part of the Isabelle distribution, http://isabelle.in.tum.de/doc/logics-HOL.pdf.

8. B. P. Mahony. The DOVE approach to the design of complex dynamic processes. 2002. *in* 'TPHOLs 2002 (Track B)', http://techreports.larc.nasa.gov/ltrs/PDF/2002/cp/NASA-2002-cp211736.pdf.

9. J. M. Spivey. *The Z Notation: A Reference Manual*. second edn, Prentice Hall International.

10. G. Smith. *A logic for object-Z*, 1994. Technical Report 94-48, Software Verification Research Center, The University of Queensland.

11. L. Wildman. Requirements reformulation using formal specification: A case study. In L. M. Kristensen C. Lakos, R. Esser and J. Billington, editors, *Proceedings of the Workshop on the use of Formal Methods in Defence Systems*, Conferences in Research and Practice in Information Technology, ACS, pages 75–83, 2002. http://crpit.com/confpapers/CRPITV12Wildman.ps.

12. C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In *MPC '98: Proceedings of the Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 188–206, London, UK, 1998. Springer-Verlag.

13. C. C. Morgan. *Programming from Specifications*. second edn, Prentice Hall International.

14. A. Finkelstein and W. Emmerich. The future of requirements management tools. In R. Wagner G. Quirchmayr and M. Wimmer, editors, *Information Systems in Public Administration and Law*, Oesterreichische Computer Gesellschaft, 2000.

DISTRIBUTION LIST

Tools for Requirements Management: a Comparison of Telelogic DOORS and the HIVE

Tony Cant, Jim McCarthy and Robyn Stanley

	Number of Copies
DEFENCE ORGANISATION	
Task Sponsor	
Ag/DPPI, DMO	5 (printed)
S&T Program	
Chief Defence Scientist	1
Deputy Chief Defence Scientist Policy	1
AS Science Corporate Management	1
Director General Science Policy Development	1
Counsellor, Defence Science, London	Doc Data Sheet
Counsellor, Defence Science, Washington	Doc Data Sheet
Scientific Adviser to MRDC, Thailand	Doc Data Sheet
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sheet and Dist List
Scientific Adviser, Army	Doc Data Sheet and Dist List
Air Force Scientific Adviser	Doc Data Sheet and Exec Summ
Scientific Adviser to the DMO	Doc Data Sheet and Dist List
Information Sciences Laboratory	
Chief, IND	1
Research Leader, IA	1
Head, TCS Group	1
Task Manager, Tony Cant	1
Author	5 (printed)
DSTO Library and Archives	
Library, Edinburgh	1 (printed)
Defence Archives	1 (printed)
Capability Development Group	
Director General Maritime Development	Doc Data Sheet
Director General Capability and Plans	Doc Data Sheet
Assistant Secretary Investment Analysis	Doc Data Sheet
Director Capability Plans and Programming	Doc Data Sheet

Director General Australian Defence Simulation Office	Doc Data Sheet
Chief Information Officer Group	
Head Information Capability Management Division	Doc Data Sheet
AS Information Strategy and Futures	Doc Data Sheet
Director General Information Services	Doc Data Sheet
Strategy Group	
Director General Military Strategy	Doc Data Sheet
Assistant Secretary Governance and Counter-Proliferation	Doc Data Sheet
Navy	
Director General Navy Capability, Performance and Plans, Navy Headquarters	Doc Data Sheet
Director General Navy Strategic Policy and Futures, Navy Headquarters	Doc Data Sheet
Deputy Director (Operations) Maritime Operational Analysis Centre, Building 89/90, Garden Island, Sydney	Doc Data Sheet and Dist List
Deputy Director (Analysis) Maritime Operational Analysis Centre, Building 89/90, Garden Island, Sydney	
Army	
ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal	Doc Data Sheet (pdf format)
SO (Science), Deployable Joint Force Headquarters (DJFHQ)(L), Enoggera QLD	Doc Data Sheet
SO (Science), Land Headquarters (LHQ), Victoria Barracks, NSW	Doc Data Sheet and Exec Summ
Air Force	
SO (Science), Headquarters Air Combat Group, RAAF Base, Williamtown	Doc Data Sheet and Exec Summ
Joint Operations Command	
Director General Joint Operations	Doc Data Sheet
Chief of Staff Headquarters Joint Operation Command	Doc Data Sheet
Commandant, ADF Warfare Centre	Doc Data Sheet
Director General Strategic Logistics	Doc Data Sheet
COS Australian Defence College	Doc Data Sheet
Intelligence and Security Group	
Assistant Secretary, Concepts, Capabilities and Resources	1
DGSTA, DIO	1
Manager, Information Centre, DIO	1
Director Advanced Capabilities, DIGO	Doc Data Sheet

Defence Materiel Organisation

Deputy CEO, DMO	1
Head Aerospace Systems Division	Doc Data Sheet
Head Maritime Systems Division	Doc Data Sheet
Program Manager Air Warfare Destroyer	Doc Data Sheet
CDR Joint Logistics Command	Doc Data Sheet
GWEO-DDP	Doc Data Sheet

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy Library	1
Head of Aerospace and Mechanical Engineering, ADFA	1
Hargrave Library, Monash University	Doc Data Sheet

OTHER ORGANISATIONS

National Library of Australia	1
NASA (Canberra)	1

INTERNATIONAL DEFENCE INFORMATION CENTRES

US - Defense Technical Information Center	1
UK - Dstl Knowledge Services	1
Canada - Defence Research Directorate R&D Knowledge and Information Management (DRDKIM)	1
NZ - Defence Information Centre	1

ABSTRACTING AND INFORMATION ORGANISATIONS

Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts, US	1
Documents Librarian, The Center for Research Libraries, US	1

INFORMATION EXCHANGE AGREEMENT PARTNERS

National Aerospace Laboratory, Japan	1
National Aerospace Laboratory, Netherlands	1

SPARES

DSTO Edinburgh Library	5 (printed)

Total number of copies: printed 17, pdf 27

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. CAVEAT/PRIVACY MARKING

2. TITLE	3. SECURITY CLASSIFICATION
Tools for Requirements Management: a Comparison of Telelogic DOORS and the HiVe	Document (U) Title (U) Abstract (U)
4. AUTHORS	5. CORPORATE AUTHOR
Tony Cant, Jim McCarthy and Robyn Stanley	Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO–GD–0466	AR 013-689	General Document	July 2006

8. FILE NUMBER	9. TASK NUMBER	10. SPONSOR	11. No OF PAGES	12. No OF REFS
	JTW 04/061	DMO	46	14

13. URL OF ELECTRONIC VERSION	14. RELEASE AUTHORITY
http://www.dsto.defence.gov.au/corporate/reports/DSTO–GD–0466.pdf	Chief, Information Networks Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved For Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DEFTEST DESCRIPTORS

Modelling	Critical Systems
System Design	Verification

19. ABSTRACT

It is now well-known that a robust and complete requirements management process is of great benefit in the procurement of complex, critical, software-intensive systems. DOORS is a well-established suite of software made by Telelogic, designed to maintain large sets of requirements. The HiVe is a project under development by the TCS Group at DSTO that aims to provide a new approach to the creation of technical documents required in system development. It can be used to formulate, manage, and analyse requirements, and then to develop the system design which satisfies them. While the main focus of each piece of software is different, there is enough overlap that users of DOORS would strongly benefit from use of the HiVe. This report highlights the strengths of both tools, compares their major features, and suggests a number of ways the HiVe and DOORS can interact with one another to benefit the user.